Formal Verification of the Binary Exponential Backoff Protocol

Jozef Hooman

Department of Mathematics and Computing Science,

Eindhoven University of Technology,

P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands.

hooman@win.tue.nl

http://www.tue.nl/~hooman

Abstract

We present a formal framework for the specification and verification of distributed real-time systems. To obtain mechanical support, this framework has been defined in the language of the proof checker PVS. Intermediate stages of the design are represented by mixed terms where specifications and programming constructs can be combined. Compositional proof rules allow the verification of design steps. Here we focus on rules for parallel composition and hiding. Their use during protocol verification is illustrated by a part of the HTTP protocol, the binary exponential backoff protocol.

1 Introduction

Distributed real-time systems usually have a large number of possible executions, making exhaustive testing impossible. To increase the confidence in the correctness of such systems, which are often safety-critical, we investigate the use of formal methods. Formal verification could be done after the complete program for the system has been written, but this will be extremely complex. Hence we aim at the verification of design steps during the process of system development. This verify-while-design paradigm allows an early detection of design errors and ideally leads to systems that are correct by construction.

Clearly, the formal verification of a realistic system requires suitable tool support. Usually there are a large number of proof obligations; although many of them are often almost trivial, they have to be checked carefully (not checking such details is a source of many errors). A tool is needed that can discharge trivial proof obligations automatically. Moreover, mechanical support is needed to keep track of the dependencies between definitions, lemmas and theorems, especially since specifications and designs are frequently changed during the development.

Formal specification of real-time systems requires a formalism in which the timed occurrence of events can be expressed. A large number of formalisms has been devised, often extending non-real-time frameworks such as temporal logic, process algebra, automata, and Hoare logic (see, for instance, [1, 10]). Here we consider a basic framework in which a component is simply specified by $spec(Eset, A)$, where $Eset$ is the *alphabet* of the component (the set of events that form the static interface) and A is an *assertion* expressing its timed behaviour. Timing is expressed in terms of a conceptual global clock of an external observer of the system.

Mixed terms are used to represent intermediate stages during system design, where parts of the system are already implemented and other parts are only specified; they allow a free mixture of specifications and programming constructs. Let $Spec_1 \Rrightarrow Spec_2$ denote that $Spec_1$ refines (i.e. implements) $Spec_2$. Then a design can be represented by a chain of subsequent refinements of mixed terms. For instance,

$Spec \Lleftarrow (Spec_1 \| Specs_2) \Lleftarrow ((Spec_{11}\, ;\, Spec_{12}) \| Specs2) \Lleftarrow ((x := 5\, ;\, Spec_{12}) \| Specs2)$, etcetera.

Formal verification of refinement steps is based on *compositional* proof rules, that is, for each compound programming construct there is a rule in which a specification for the construct can be derived using only the *specifications* of the components, without knowing their implementation. Hence components can be considered as black boxes, which can be implemented later according to their specification. As an example, consider the parallel composition rule which is especially important during the early stages of system development. We will show that we can obtain a sound rule of the following form, expressing that parallel composition corresponds to the conjunction of specifications, provided some conditions on the specifications hold.

$$\frac{\text{conditions on specifications}}{spec(Eset_1, A_1) \| spec(Eset_2, A_2) \Rrightarrow spec(Eset_1 \cup Eset_2, A_1 \wedge A_2)}$$

Moreover, the refinement chain above shows that it should be possible to perform refinements in a certain context. For instance, from $Spec_1 \Lleftarrow (Spec_{11}\, ;\, Spec_{12})$ we should be able to derive $(Spec_1 \| Specs_2) \Lleftarrow ((Spec_{11}\, ;\, Spec_{12}) \| Specs2)$. This is formalized in so-called monotonicity rules. For instance, for parallel composition we have

$$\frac{comp_1 \Rrightarrow comp_3, \quad comp_2 \Rrightarrow comp_4}{comp_1 \parallel comp_2 \Rrightarrow comp_3 \parallel comp_4}$$

Similar rules can be given for other programming constructs. Soundness of these proof rules is based on a denotational semantics which expresses the timed behaviour of programming constructs. In this paper we focus on parallel composition and hiding. The hiding construct can be used to encapsulate internal events.

As an application of our formal framework, we present a general approach to verify distributed real-time protocols. This is illustrated by the verification of a Binary Exponential Backoff algorithm, which is part of the HTTP protocol [2].

Tool support for our mixed framework is obtained by formulating it in the language of the verification system PVS (Prototype Verification System) [13, 14]. The language of PVS is based on typed higher-order logic and contains a large number of pre-defined types. Typechecking is not decidable, but might generate proof obligations (so-called Type Check Conditions). Moreover PVS contains an interactive theorem prover which contains powerful decision procedures to prove simple properties automatically. In general, the user can prove a theorem by applying commands that simplify the goal until it can be proved automatically.

We do not only use PVS for the verification of concrete applications, but also for the development of a theoretical framework. For instance, to obtain mechanically checked soundness proofs of the verification rules. Further, we frequently use so-called "putative" theorems, that is, we try to prove properties that ought to hold. A failure to prove such a property reveals errors, whereas a successful proof increases the confidence in the formalization. All theorems and lemmas presented here have been proved in PVS.

The remainder of this paper is structured as follows. Primitives to reason about real-time properties are defined in section 2. Section 3 contains the basic semantic framework to express timing behaviour of components. The timed semantics of parallel composition and hiding is defined in section 4. Verification rules for these constructs can be found in section 5. They are used in a general approach to protocol verification, as formulated in section 6. In section 7 this approach is applied to the Binary Exponential Backoff protocol. Finally, section 8 contains a few concluding remarks.

2 Real-Time Framework

The verification system PVS provides a very general typed higher-order logic and allows structuring and modularization by means of parameterized theories. To able to deal with a particular class of applications, the user has to define its own framework in this logic. Since we want to describe the behaviour of real-time systems, first a number of timing primitives are introduced in a general theory *TimePrim*.

This theory has three parameters, a type *Time* and two orders on this type, using the pre-defined predicates *strict_order?* and *partial_order?*. The theory contains an assumption about the usual relation between the two orders. If a theory imports TimePrim with a particular time domain and certain orders, PVS generates Type Check Conditions requiring that the orders are indeed strict and total, resp., and property *leq_less* holds. Of course, we could also define $\leq$ by $<$, but we have chosen for the assuming clause to be able to exploit the pre-defined types and orders of PVS, and thus make optimal use of the associated decision procedures of PVS.

The standard boolean connectives are overloaded for predicates on Time, i.e. functions from Time to bool. Intervals are considered as a set of time points, which are in PVS also represented as functions from Time to bool. As indicated by the dots "...", we only show some of the definitions.

Next we define when a predicate holds *inside* or *during* an interval. As an example, we show a simple lemma named *dur_inside* concerning the relation between these primitives. It can be proved automatically by the PVS proof checker.

TimePrim[Time : TYPE, $<$: (strict_order?[Time]), $\leq$: (partial_order?[Time])] : THEORY

BEGIN

ASSUMING

leq_less : ASSUMPTION $\forall\ (t_1, t_2 :\ \text{Time}) :\ t_1 \leq t_2 \Leftrightarrow t_1 < t_2 \vee t_1 = t_2$

ENDASSUMING

$t, t_0, t_1, t_2 :$ VAR Time

$P, Q :$ VAR pred[Time]

$\neg P :\ \text{pred[Time]}\ =\ \lambda\ t :\ \neg P(t);$

$P \wedge Q : \text{ pred[Time]} = \lambda\, t : P(t) \wedge Q(t);$

$\ldots$

$[t_0, t_1] : \text{ setof[Time]} = \{t \mid t_0 \leq t \wedge t \leq t_1\}$

$[t_0, t_1) : \text{ setof[Time]} = \{t \mid t_0 \leq t \wedge t < t_1\}$

$\ldots$

$I : \text{ VAR setof[Time]}$

$P \textbf{ in } I : \text{ bool} = \exists\, t : t \in I \wedge P(t)$

$P \textbf{ during } I : \text{ bool} = \forall\, t : t \in I \Rightarrow P(t)$

dur_inside : LEMMA $\neg(P \textbf{ in } I) \Leftrightarrow (\neg P) \textbf{ during } I$

END TimePrim

Henceforth we do not repeat declarations of variables, using for instance $t, t_0, t_1, \ldots$ as variables over Time. Moreover, we omit the structure of theory names and imported theories, but focus on the main ideas. For more details about the formal framework we refer to [7].

3 Semantic Primitives

Program semantics is defined in terms of the events that can be observed at any point of time. For simplicity, to emphasize the main concepts, we do not consider the local state of a component here. More details on the incorporation of a local state can be found in [8].

Events are represented by a non-empty type; particular events can later be defined as constants of this type. Basic primitive is an *observation function*, which assigns to each point in time the set of events that occur at that time. Operations on sets are overloaded to operations on observation functions.

Events : NONEMPTY_TYPE

ObsFuncts : TYPE $= [\text{Time} \rightarrow \text{setof[Events]}]$

$o, o_1, o_2 :$ VAR ObsFuncts

$\text{Eset}: \ \text{VAR setof[Events]}$

$$o_1 \cup o_2 : \ \text{ObsFuncts} \ = \ \lambda\ t: \ o_1(t) \cup o_2(t)$$
$$o \cap \text{Eset}: \ \text{ObsFuncts} \ = \ \lambda\ t: \ o(t) \cap \text{Eset}$$
$$o \setminus \text{Eset}: \ \text{ObsFuncts} \ = \ \lambda\ t: \ o(t) \setminus \text{Eset}$$
$$o \subseteq \text{Eset}: \ \text{bool} \ = \ \forall\ t: \ o(t) \subseteq \text{Eset}$$

The basic structure to describe program components is given by type *CompInfo* consisting of records with two fields: α represents the alphabet of the component and *obs* describes the possible behaviours by a predicate on observation functions. The type *Components* requires that an observation of a component contains only events of its alphabet.

$$\begin{aligned}\text{CompInfo}: \ \text{TYPE} \ = \ [\#\ \alpha \ &: \ \text{setof[Events]},\\ \text{obs} &: \ \text{pred[ObsFuncts]}\ \#]\end{aligned}$$

$\text{ci}: \ \text{VAR CompInfo}$

$$\text{CompProp(ci)}: \ \text{bool} \ = \ \forall\ o: \ \text{obs(ci)}(o) \Rightarrow o \subseteq \alpha(\text{ci})$$

$$\text{Components}: \ \text{TYPE} \ = \ \{\text{ci} \mid \ \text{CompProp(ci)}\}$$

Component comp1 *refines* component comp2, denoted by comp1 $\Rrightarrow$ comp2, if the alphabet of "specification" comp2 is contained in that of "implementation" comp1 and any behaviour of comp1 is also one of comp2. The refinement relation is reflexive and transitive.

$\text{comp, comp0, comp1, comp2, comp3}: \ \text{VAR Components}$

$$\text{comp1} \Rrightarrow \text{comp2}: \ \text{bool} \ = \alpha(\text{comp2}) \subseteq \alpha(\text{comp1}) \wedge \text{obs(comp1)} \subseteq \text{obs(comp2)}$$

$$\text{RefRefl}: \ \text{THEOREM comp} \Rrightarrow \text{comp}$$

$$\begin{aligned}\text{RefTrans}: \ \text{THEOREM} \ &(\text{comp0} \Rrightarrow \text{comp2}) \Leftrightarrow\\ &(\exists\ \text{comp1}: \ (\text{comp0} \Rrightarrow \text{comp1}) \wedge (\text{comp1} \Rrightarrow \text{comp2}))\end{aligned}$$

Specifications consist of an alphabet and an assertion, i.e., a predicate on observation functions. To obtain a framework of mixed terms, a specifications is also of type Components. An observation of a specification should satisfy the assertion and additionally contain only events of the alphabet (to obtain CompProp). For simplicity, specifications contain a single assertion here, but the framework can be extended easily to a framework with, for instance, pre and post conditions.

$$\text{Assertion}: \ \text{TYPE} \ = \ \text{pred}[\text{ObsFuncts}]$$

$$A, A_1, A_2: \ \text{VAR Assertion}$$

$$\text{Valid}(A): \ \text{bool} \ = \ \forall\, o: \ A(o)$$

$$\begin{aligned} \text{spec}(\text{Eset}, A): \ \text{Components} \ = \ (\#\ \alpha \ &:= \ \text{Eset}, \\ \text{obs} &:= \ \lambda\, o: \ o \subseteq \text{Eset} \wedge A(o)\ \#) \end{aligned}$$

Observe that $o(t)(\text{read})$ expresses that event *read* occurs at time t in observation o. It is often convenient to be able to write $o(\text{read})(t)$ and then, using the primitives of TimePrim, also $o(\text{read})$ **in** $[2, 7]$. This is achieved by means of a conversion.

$$E: \ \text{VAR Events}$$

$$\text{At}(o)(E)(t): \ \text{bool} \ = \ o(t)(E)$$

$$\text{CONVERSION At}$$

This means that an occurrence of $o(E)(t)$ is interpreted as $\text{At}(o)(E)(t)$, i.e. $o(t)(E)$.

4 Semantics of Parallel Composition and Hiding

In this section a denotational semantics of parallel composition and hiding is defined.

4.1 Semantics of Parallel Composition

The denotational semantics of the parallel composition of two components is defined in terms of the semantics of these components. For the alphabet we simply take the union of the alphabets of the components. Hiding of internal events is considered as a seperate operation, defined in the next section. Inspired by the trace-based untimed semantics of parallel composition [3], we define the timed behaviour by means of so-called projections, here represented by intersection. Main requirement is that the projection of an observation of the parallel composition onto the alphabet of one component, should lead to an observation of this component. Additionally, observations should only contain events of the alphabet.

$$
\begin{aligned}
&//(\mathrm{comp1}, \mathrm{comp2}) : \ \mathrm{Comps} \ = \\
&\quad (\#\ \alpha := \ \alpha(\mathrm{comp1}) \cup \alpha(\mathrm{comp2}), \\
&\qquad \mathrm{obs} := \lambda\ o : (\exists\ o_1, o_2 : \mathrm{obs}(\mathrm{comp1})(o_1) \wedge \mathrm{obs}(\mathrm{comp2})(o_2) \wedge \\
&\qquad\qquad o \cap \alpha(\mathrm{comp1}) = o_1 \wedge o \cap \alpha(\mathrm{comp2}) = o_2 \wedge \\
&\qquad\qquad o \subseteq \alpha(\mathrm{comp1}) \cup \alpha(\mathrm{comp2}))\ \#)
\end{aligned}
$$

The definition above is convenient for the soundness proof of the verification rule that will be presented in the next section; other equivalent versions, e.g. using the intersection of behaviours, can be found in [7]. We have proved that parallel composition is commutative and associative.

$$\mathrm{ParComm}: \ \textsc{lemma}\ \mathrm{comp1}\ //\ \mathrm{comp2} = \mathrm{comp2}\ //\ \mathrm{comp1}$$

$$\mathrm{ParAssoc}: \ \textsc{lemma}\ (\mathrm{comp1}\ //\ \mathrm{comp2})\ //\ \mathrm{comp3} = \mathrm{comp1}\ //\ (\mathrm{comp2}\ //\ \mathrm{comp3})$$

4.2 Semantics of Hiding

The hiding (or encapsulation) construct *comp - Eset* hides the events of set *Eset* from component *comp*. Clearly, this means that these events are removed from the alphabet. Moreover, the events from *Eset* are removed from the observations of *comp*.

$$
\begin{aligned}
&-(\mathrm{comp}, \mathrm{Eset}) : \ \mathrm{Comps} \ = \\
&\quad (\#\ \alpha \ := \ \alpha(\mathrm{comp}) \setminus \mathrm{Eset}, \\
&\qquad \mathrm{obs} := \ \lambda\ o : \ (\exists\ o_1 : \ \mathrm{obs}(\mathrm{comp})(o_1) \wedge o = o_1 \setminus \mathrm{Eset})\ \#)
\end{aligned}
$$

5 Verification Rules

We present the consequence rule and rules for parallel composition and hiding. Soundness of these rules has been proved in PVS on the basis of the semantics of the previous section.

5.1 Consequence Rule

The consquence rule allows us to weaken assertions.

$$\begin{aligned}\text{ConsRule}: \ \textsc{theorem}\ & \text{Eset0} = \text{Eset} \wedge \text{Valid}(A_0 \Rightarrow A) \Rightarrow \\ & (\text{spec}(\text{Eset0}, A_0) \Rrightarrow \text{spec}(\text{Eset}, A))\end{aligned}$$

5.2 Rules for Parallel Composition.

It is rather easy to show the next monotonicity property which makes it possible to perform refinements in a parallel context.

$$\begin{aligned}\text{MonoPar}: \ \textsc{theorem}\ & (\text{comp1} \Rrightarrow \text{comp3}) \wedge (\text{comp2} \Rrightarrow \text{comp4}) \Rightarrow \\ & (\text{comp1} \parallel \text{comp2} \Rrightarrow \text{comp3} \parallel \text{comp4})\end{aligned}$$

The next rule for parallel composition essentially expresses that the parallel composition of specifications corresponds to the conjunction of assertions. To achieve a sound rule, it is required that the validity of the assertion of a component depends only on its alphabet, as expressed by predicate *OnlyDepEve*.

$$\text{OnlyDepEve}(A, \text{Eset}): \ \text{bool} \ = \forall\ o: \ A(o) \Leftrightarrow A(o \cap \text{Eset})$$

$$\begin{aligned}\text{ParCompRule}: \ \textsc{theorem}\ & \text{OnlyDepEve}(A_1, \text{Eset1}) \wedge \\ & \text{OnlyDepEve}(A_2, \text{Eset2}) \Rightarrow \\ & \text{spec}(\text{Eset1}, A_1) \parallel \text{spec}(\text{Eset2}, A_2) \Rrightarrow \text{spec}(\text{Eset1} \cup \text{Eset2}, A_1 \wedge A_2)\end{aligned}$$

Soundness of the parallel composition rule, i.e. theorem *ParCompRule*, has been proved along the following lines. Assume $\text{OnlyDepEve}(A_i, \text{Eseti})$, for $i = 1, 2$.

First observe that the alphabets of the components on both sides of the refinement are equal.

Next consider an observation $o \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset1}, A_1) \parallel \mathrm{spec}(\mathrm{Eset2}, A_2))$. Let $i \in \{1, 2\}$. By the definition of the semantics there exists o_i such that $o \cap \mathrm{Eseti} = o_i$ and $\mathrm{obs}(\mathrm{spec}(\mathrm{Eseti}, A_i))(o_i)$, thus $A_i(o_i)$. Hence $A_i(o \cap \mathrm{Eseti})$. Then OnlyDepEve leads to $A_i(o)$. Since $o \subseteq \mathrm{Eset1} \cup \mathrm{Eset2}$, this leads to $o \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset1} \cup \mathrm{Eset2}, A_1 \wedge A_2))$.

5.3 Proof Rules for Hiding

For the hiding construct we also show a monotonicity property expressed by theorem *HideMono*. Next we prove a rule which expresses that a set of events can be hidden by simply removing it from the alphabet, provided the assertion does not depend on the events removed, i.e. only depends on the resulting events.

$$\mathrm{HideMono}: \ \textsc{theorem}\ (\mathrm{comp1} \Rrightarrow \mathrm{comp2}) \Rightarrow (\mathrm{comp1} - \mathrm{Eset} \Rrightarrow \mathrm{comp2} - \mathrm{Eset})$$

$$\begin{aligned} \mathrm{HideRule}: \ \textsc{theorem}\ & \mathrm{OnlyDepEve}(A, \mathrm{Eset} \setminus \mathrm{Eset0}) \Rightarrow \\ & \qquad \mathrm{spec}(\mathrm{Eset}, A) - \mathrm{Eset0} \Rrightarrow \mathrm{spec}(\mathrm{Eset} \setminus \mathrm{Eset0}, A) \end{aligned}$$

Soundness of this rule is proved as follows. Assume $\mathrm{OnlyDepEve}(A, \mathrm{Eset} \setminus \mathrm{Eset0})$.
First observe that $\alpha(\mathrm{spec}(\mathrm{Eset}, A) - \mathrm{Eset0}) = \alpha(\mathrm{spec}(\mathrm{Eset} \setminus \mathrm{Eset0}, A))$.
Next consider $o \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset}, A) - \mathrm{Eset0})$. Hence there exists an o_1 such that $o = o_1 \setminus \mathrm{Eset0}$ and $o_1 \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset}, A))$, thus $o_1 \subseteq \mathrm{Eset}$ and $A(o_1)$. By $\mathrm{OnlyDepEve}(A, \mathrm{Eset} \setminus \mathrm{Eset0})$ we obtain $A(o_1 \cap \mathrm{Eset} \setminus \mathrm{Eset0})$. Since $o_1 \subseteq \mathrm{Eset}$ this leads to $A(o_1 \setminus \mathrm{Eset0})$ and hence $A(o)$. Since $o \subseteq \mathrm{Eset} \setminus \mathrm{Eset0}$, we obtain $o \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset} \setminus \mathrm{Eset0}, A))$.

6 Application to Protocol Verification

The formal framework is rather general general, intended for a wide range of applications. It is convenient to have some guidelines for a particular class of applications. Here we consider the verification of distributed real-time protocols and formulate a few simple steps for a network of nodes.

1. Model the application domain, that is, describe the main primitives that are needed to express the service specification.

2. Specify the service to be provided by the network.

3. Specify the communication mechanism between nodes.

4. Specify the protocol performed by each node, in terms of its alphabet only.

5. Verify the protocol using the parallel composition rule (and the consequence rule), i.e., prove that the specifications of the nodes (point 4) and the communication mechanism (point 3) lead to the required service (point 2). Internal events can be removed by the hiding rule.

7 Binary Exponential Backoff Protocol

As an example of the approach of the previous section, we consider the verification of the Binary Exponential Backoff Protocol, a small part of the HyperText Transfer Protocol HTTP/1.1. This protocol is a proposed standard, described in RFC 2068 [2]. It organizes the transfer of data between a client (e.g. a network browser) and a server. Client and server communicate by means of a TCP connection.

Document RFC 2068 describes only the client part of the protocol, along the following lines.

1. Initiate a new connection to the server;

2. transmit request headers;

3. initialize R to the round-trip time (if known) or 5 seconds;

4. compute $T = R^N$, with N the number of retries;

5. wait for an error or T seconds;

6. if no error has been received after T seconds, start transmitting data;

7. if the connection is aborted, repeat step 1.

Given this description, figure 1 shows the events that are relevant for the Binary Exponential Backoff protocol.

RFC 2068 is unclear about the purpose and the context of this protocol, but a domain expert provided some useful hints; the idea is that the client wants to send data to the server

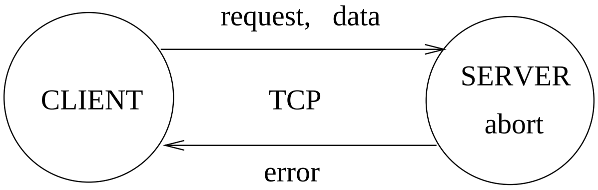

Figure 1: Client and server of the HTTP protocol

and requests permission to do this. The request, however, might be invalid, e.g. because the authorization code is not correct, and the server sends an error message (instead of a normal response). If the error is delayed, the client decides to start sending the data already (step 6 above). When the server gets too much unwanted data, it aborts the connection and then also the error message might get lost. Consequently, the client will try again, with a new time-out value. The protocol is intended to guarantee that the client eventually receives the error.

Our aim is to investigate the essential functioning of the protocol on an abstract level, to achieve better understanding. This includes relations between the timing of components and, for instance, the value of the time-out of the client.

7.1 Model the application domain

As a time domain we take the reals with the standard orderings, which are already available in PVS. Our real-time framework is incorporated by importing theory TimePrim. Moreover, the framework of the previous sections is imported, although this is not shown here. Messages and nodes are defined as enumeration types, which implies that all elements are different, a fact that is used by the decision procedures of PVS. A *send* and a *receive* of a message by a node and an *abort* are declared as events. Moreover, assume given a predicate that expresses which messages are erroneous.

$$\mathrm{Time}: \textsc{type} \;=\; \mathrm{real}$$
$$\mathrm{NonNegTime}: \textsc{type} \;=\; \{t : \mathrm{Time} \mid t \geq 0\}$$

$$\textsc{importing}\ \mathrm{TimePrim}[\mathrm{Time}, <, \leq]\ , \ldots$$

$$\mathrm{Messages}: \textsc{type} \;=\; \{\mathrm{request}, \mathrm{data}, \mathrm{error}\}$$

$m, m_1, m_2 :$ VAR Messages

Nodes : TYPE $= \{\text{client}, \text{server}\}$

node, node1, node2 : VAR Nodes

send(node, m) : Events

rec(node, m) : Events

abort : Events

erroneous(m) : bool

7.2 Specify the required service

The service specification expresses that if the client sends an erroneous request, it receives an error between lower bound L and upper bound U. These time bounds have been added to be able to investigate the timing relations between components.

$L, U :$ Time

$$\text{ETL} : \text{setof[Events]} = \{\text{E} \mid \text{E} = \text{send}(\text{client}, \text{request}) \vee \text{E} = \text{rec}(\text{client}, \text{error})\}$$

$$\text{ATL} : \text{Assertion} = \lambda\ o : \forall\ t : o(\text{send}(\text{client}, \text{request}))(t) \wedge \text{erroneous}(\text{request}) \Rightarrow o(\text{rec}(\text{client}, \text{error}))\ \mathbf{in}\ [t + L, t + U]$$

$$\text{TLSpec} : \text{Components} = \text{spec}(\text{ETL}, \text{ATL})$$

7.3 Specify the communication mechanism

Next we axiomatize the relation between send, receive, and abort events. Only two properties of the underlying TCP protocol are needed. First we specify that a message send will be received between certain time bounds, provided no abort event takes place.

TransDelayL, TransDelayU : NonNegTime

$$\mathrm{SendRec}: \ \textsc{axiom}\ o(\mathrm{send}(\mathrm{node1}, m))(t) \wedge \neg o(\mathrm{abort})\ \mathbf{during}\ [t, t + \mathrm{TransDelayU}] \Rightarrow$$
$$\forall(\mathrm{node2} \mid \ \mathrm{node2} \neq \mathrm{node1}):$$
$$o(\mathrm{rec}(\mathrm{node2}, m))\ \mathbf{in}\ [t + \mathrm{TransDelayL}, t + \mathrm{TransDelayU}]$$

Moreover, we express that a message is only received if it has been sent between certain time bounds.

$$\mathrm{RecSend}: \ \textsc{axiom}\ o(\mathrm{rec}(\mathrm{node1}, m))(t) \Rightarrow$$
$$\exists t_0, \mathrm{node2} : \mathrm{node2} \neq \mathrm{node1} \wedge$$
$$t_0 \in [t - \mathrm{TransDelayU}, t - \mathrm{TransDelayL}] \wedge o(\mathrm{send}(\mathrm{node2}, m))(t_0)$$

The time bounds mentioned here are not present in the TCP protocol, but have been added to be able to reason about the transmission speed.

7.4 Specify the protocol performed by the nodes

Next the protocol performed by the nodes is specified; here we have to specify the server and the client.

7.4.1 Specify protocol of the server

The server should send an error message, between certain time bounds, if an erroneous request is received.

$$\mathrm{ErrL}, \mathrm{ErrU}: \ \mathrm{NonNegTime}$$

$$\mathrm{AS1}: \ \mathrm{Assertion} \ = \lambda\ o : \forall\ t : o(\mathrm{rec}(\mathrm{server}, \mathrm{request}))(t) \wedge \mathrm{erroneous}(\mathrm{request}) \Rightarrow$$
$$o(\mathrm{send}(\mathrm{server}, \mathrm{error}))\ \mathbf{in}\ [t + \mathrm{ErrL}, t + \mathrm{ErrU}]$$

Main complication of the protocol is that the server might close the connection by an abort when it receives too much unwanted data. For the correctness it is, however, important that this is the only reason for the server to abort the connection. So when it does an abort, some data must have been received recently, i.e. between certain time bounds.

$$\mathrm{AbortL}, \mathrm{AbortU} : \mathrm{NonNegTime}$$

$$\mathrm{AS2} : \mathrm{Assertion} = \lambda\, o : \forall\, t : o(\mathrm{abort})(t) \Rightarrow$$
$$\exists t_0 : t_0 \in [t - \mathrm{AbortU}, t - \mathrm{AbortL}] \wedge o(\mathrm{rec}(\mathrm{server}, \mathrm{data}))(t_0)$$

$$\mathrm{AS} : \mathrm{Assertion} = \mathrm{AS1} \wedge \mathrm{AS2}$$

$$\mathrm{ES} : \mathrm{setof[Events]} = \{\mathrm{E} \mid \mathrm{E} = \mathrm{rec}(\mathrm{server}, \mathrm{request}) \vee \mathrm{E} = \mathrm{rec}(\mathrm{server}, \mathrm{data}) \vee$$
$$\mathrm{E} = \mathrm{send}(\mathrm{server}, \mathrm{error}) \vee \mathrm{E} = \mathrm{abort}\}$$

$$\mathrm{Server} : \mathrm{Components} = \mathrm{spec}(\mathrm{ES}, \mathrm{AS})$$

7.4.2 Specify protocol of the client

To get insight in the main principles of the protocol, we abstract from the algorithm of the client that dynamically computes the distance between a request and subsequent data. Essential is only that there is sufficient time between a request and subsequent data to allow the error to reach the client. Hence, as a first attempt the client has been specified by the following formula.

$$o(\mathrm{send}(\mathrm{client}, \mathrm{request}))(t) \Rightarrow \neg o(\mathrm{send}(\mathrm{client}, \mathrm{data}))\ \mathbf{during}\ [t, t + \mathrm{NoDataAfterPeriod}]$$

An attempt to verify this protocol revealed that this assertion is too weak; a request might be processed very fast (by TCP and server), whereas an old data message, sent before the request, might be processed very slow, generating a disturbing abort. Hence, the specification is modified to include also a period before the request.

$$\mathrm{NoDataBeforePeriod},\ \mathrm{NoDataAfterPeriod} : \mathrm{NonNegTime}$$

$$\mathrm{AC} : \mathrm{Assertion} = \lambda\, o : \forall\, t : o(\mathrm{send}(\mathrm{client}, \mathrm{request}))(t) \Rightarrow$$
$$\neg o(\mathrm{send}(\mathrm{client}, \mathrm{data}))\ \mathbf{during}\ [t - \mathrm{NoDataBeforePeriod}, t + \mathrm{NoDataAfterPeriod}]$$

$$\mathrm{EC} : \mathrm{setof[Events]} = \{\mathrm{E} \mid \mathrm{E} = \mathrm{send}(\mathrm{client}, \mathrm{request}) \vee$$
$$\mathrm{E} = \mathrm{send}(\mathrm{client}, \mathrm{data}) \vee \mathrm{E} = \mathrm{rec}(\mathrm{client}, \mathrm{error})\}$$

$$\text{Client}: \text{Components} = \text{spec}(\text{EC}, \text{AC})$$

7.5 Verification of the protocol

To verify the protocol, we first prove that a client request implies that no abort occurs in the next D time units, provided certain conditions hold (as explained below).

$$\begin{aligned}\text{ReqAbort}: \textsc{lemma}\ & \text{NoDataBeforePeriod} \geq \text{TransDelayU} + \text{AbortU} \wedge \\ & \text{NoDataAfterPeriod} \geq D - \text{TransDelayL} - \text{AbortL} \wedge \\ & \text{AS2}(o) \wedge \text{AC}(o) \Rightarrow \\ & \ (\forall\, t : o(\text{send}(\text{client}, \text{request}))(t) \Rightarrow \neg o(\text{abort})\ \textbf{during}\ [t, t + D])\end{aligned}$$

Lemma *ReqAbort* requires that *NoDataBeforePeriod* is greater than the slowest processed data, that is, a maximal TCP delay *TransDelayU* plus the upper bound of the server on performing an abort, *AbortU*. Similarly, *NoDataAfterPeriod* should be greater than the required upper bound D minus the fastest transmission delay *TransDelayL* and the fastest response by the server, *AbortL*.

For the correctness of the protocol it is required that no abort is generated during the maximal time needed to transmit an error, which equals $2 \times \mathit{TransDelayU} + \mathit{ErrU}$. Hence the D in the lemma above is replaced by this expression. Moreover, this expression determines upper bound U of the service specification. Similarly, lower bound L is determined by the fastest transmission. This leads to the following timing constraints which allow us to prove that the assertions of the server and the client lead to the required specification, as expressed in lemma *ATLLem*.

$$\begin{aligned}&\text{TimingConstraints}: \text{bool} = \\ &\quad L \leq \text{ErrL} + 2 \times \text{TransDelayL} \wedge \\ &\quad U \geq \text{ErrU} + 2 \times \text{TransDelayU} \wedge \\ &\quad \text{NoDataBeforePeriod} \geq \text{TransDelayU} + \text{AbortU} \wedge \\ &\quad \text{NoDataAfterPeriod} \geq \text{ErrU} + 2 \times \text{TransDelayU} - \text{TransDelayL} - \text{AbortL}\end{aligned}$$

$$\text{ATLLem}: \textsc{lemma}\ \text{TimingConstraints} \Rightarrow \text{Valid}(\text{AS} \wedge \text{AC} \Rightarrow \text{ATL})$$

To apply the parallel composition rule we first prove that the specifications of server and client only depend on their alphabet. Further observe that we do not yet obtain the alphabet of the service specification, but still have some additional events, represented by *IntEve*.

$$\mathrm{ASEveLem}: \ \textsc{fact}\ \mathrm{OnlyDepEve(AS, ES)}$$

$$\mathrm{ACEveLem}: \ \textsc{fact}\ \mathrm{OnlyDepEve(AC, EC)}$$

$$\mathrm{IntEve}: \ \mathrm{setof[Events]} \ = \{\mathrm{E} \mid \mathrm{E} = \mathrm{send(client, data)} \vee \mathrm{E} = \mathrm{rec(server, request)} \vee \cdots\}$$

$$\begin{aligned}\mathrm{TLPar}: \ \textsc{theorem}\ & \mathrm{TimingConstraints} \Rightarrow \\ & (\mathrm{Client} \parallel \mathrm{Server} \Rrightarrow \mathrm{spec(ETL} \cup \mathrm{IntEve, ATL))}\end{aligned}$$

Next the additional events of IntEve can be removed by the hiding rule, provided they are different from the other events. This is expressed by axiom *CommEventsDiffer*.

$$\mathrm{ATLEveLem}: \ \textsc{fact}\ \mathrm{OnlyDepEve(ATL, ETL)}$$

$$\begin{aligned}\mathrm{CommEventsDiffer}: \ \textsc{axiom}\ & \mathrm{send(node1}, m_1) \neq \mathrm{rec(node2}, m_2) \wedge \\ & \mathrm{send(node}, m) \neq \mathrm{abort} \wedge \cdots\end{aligned}$$

$$\begin{aligned}\mathrm{TLCor}: \ \textsc{theorem}\ & \mathrm{TimingConstraints} \Rightarrow \\ & ((\mathrm{Client} \parallel \mathrm{Server}) - \mathrm{IntEve} \Rrightarrow \mathrm{TLSpec})\end{aligned}$$

8 Concluding Remarks

A general framework for the formal specification and mechanical verification of distributed real-time systems has been presented. It can be considered as an extension and a modification of mixed frameworks for untimed systems [11, 12, 15]. Alternatives for the semantics and an application to hybrid systems can be found in [7]. The treatment of parallel composition for components with a local state has been studied in [8].

The formalism has been applied to a distributed real-time protocol, the Binary Exponential Backoff protocol. It has been verified on an abstract level, abstracting for instance from the algorithm which is used to compute the distance in time between request and subsequent data dynamically. This computation could be considered in a continuation of this work, where server and client can be refined and implemented. (Note that correctness of the client protocol described in section 7 requires $R > 1$.) A related refinement can be found in the work on a distributed real-time arbitration protocol, where first the protocol is verified on an abstract level, and next the nodes are implemented in isolation according to their specification [4].

Other applications of our approach to protocol verification in PVS concern part of the ACCESS.bus protocol [5] and a membership protocol, with a dynamically changing network and local clocks [6].

In [9] an alternative approach has been applied to the specification and verification of the link layer of the serial bus protocol P1394. Since the informal specification is based on communicating state machines, this framework has been formalized in PVS. The intention was to stay close to the informal text, motivated by the importance of the step from informal to formal specification. This is also a topic of current work on requirements engineering.

Acknowledgments

Many thanks goes to Koen Holtman for his suggestion to verify the binary exponential backoff protocol and his explanation of context and purpose of the protocol. Further I would like to thank Leendert Pieter van Drimmelen, Arnaud Gouder de Beauregard, Ronald Marcelis, and Alex Vrijsen for their work on this protocol as an exercise for a post-graduate course on PVS. Although their verification attempts were different from the solution presented here, their insights certainly contributed to the solution presented here.

References

[1] de Bakker, Huizing, de Roever, and Rozenberg, editors. *Proceedings REX Workshop on Real-Time: Theory in Practice*. LNCS 600. Springer-Verlag, 1992.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frysryk, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Request for Comments (RFC) 2068, http://ds.internic.net/ds/rfc-index.html, 1997.

[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[4] J. Hooman. Compositional verification of a distributed real-time arbitration protocol. *Real-Time Systems*, 6(2):173–205, 1994.

[5] J. Hooman. Verifying part of the ACCESS.bus protocol using PVS. In *Proceedings 15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 96–110. LNCS 1026, Springer-Verlag, 1995.

[6] J. Hooman. Verification of distributed real-time and fault-tolerant protocols. In *Algebraic Methodology and Software Technology (AMAST'97)*, pages 261–275. LNCS 1349, Springer-Verlag, 1997.

[7] J. Hooman. Compositional verification of real-time applications. In *Proceedings COMPOS '97, Compositionality – The Significant Difference*, to appear. LNCS, Springer-Verlag, 1998.

[8] J. Hooman. Developing proof rules for distributed real-time systems with PVS. In *Proceedings of the Workshop on Tool Support for System Development and Verification*, to appear, 1998.

[9] L. Kühne, J. Hooman, and W.P. de Roever. Towards mechanical verification of parts of the IEEE P1394 serial bus. In I. Lovrek, editor, *2nd International Workshop on Applied Formal Methods in System Design*, pages 73–85. University of Zagreb, Faculty of Electrical Engineering and Computing, 1997.

[10] H. Langmaack, W.-P. de Roever, and J. Vytopil, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*. LNCS 863. Springer-Verlag, 1994.

[11] E.-R. Olderog. Process theory: Semantics, specification and verification. In *Current Trends in Concurrency*, pages 442–509. LNCS 224, Springer-Verlag, 1985.

[12] E.-R. Olderog. *Nets, Terms and Formulas*, volume 23 of *Camebridge Tracts in Theoretical Computer Science*. Camebridge University Press, 1991.

[13] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[14] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[15] J. Zwiers. *Compositionality, Concurrency and Partial Correctness.* LNCS 321, Springer-Verlag, 1989.